Assignment 3

Software Development 2020

Department of Computer Science
University of Copenhagen

Mahmood Mohammed Seoud [TBC115] <tbc115@alumni.di.ku>>

February 25, 2022

Introduction

In this report I will cover the main elements of my third assignment. The overall goal is to expand my knowledge of object-oriented design as well as use of properties and methods. I am to implement the missing parts of TicTacToe and to test it thorougly. The given exercise expands my know-how in object-oriented programming in C# Aswell as Visual Studio Code and my .NET Core experince. I am using the latest version of C# 10.0 with .NET Core 6.0. The given assignment revovles around working on the incomplete program TicTacToe. To do this, I were to focus on approaching it with a test-driven approach.

User's guide

To run the TicTacToe program you would have to navigate to the folder by using your terminal prompt. Then you simply run the code with "dotnet run". Once started you will notice a board-frame showing up in the console. You can control the cursor within the board frame in two ways. Player one controls the cursor with "W,A,S,D" and player two controls the cursor with "I,J,K,L". Both players mark their play with space. From there on, it is regular TicTacToe.

Implementation

While coding within Tictactoe, I encountered several problems. Some had to do with the cursor and others had to do with the game logic.

The first problem I faced withing the code, was to get the cursor to move within the board only. For this problem, I decided to go with a test-driven approach. 5 functions had to be implemented. They go as follows; `MoveUp` , `MoveDown` , `MoveLeft` , `MoveRight` , `MoveCursor` . With these 5 functions, their were 4 tests respectively.

MoveUp & MoveDown

A key thing to remember, is that up and down within a computer is not as intuitive as you would think. That is because, origo (0,0) is placed at top left corner instead of bottom left. This means that to move up, you would have to decrement the Y. The same for moving down respectively. All the testing within this test file, has a setup. This makes testing so much easier, since it gets instantiated for every test. The setup has the cursor move within the middle of the board, meaning (1,1). I started out with writing a small test, foreach move as I finished testing one I moved to the next. The first one is `MoveUp` . Here A.1 I used the knowledge about how up and down works. At line 52 in A.1, I made sure that the cursor, could move all the way up to the last field, with the $>=$ logic. If the move up is a valid one, then I let the cursor move. This way, I make sure that the cursor stays within the board, while acting like a real cursor.

MoveLeft & MoveRight

The next two methods `MoveLeft` & `MoveRight` Were constructed in a similar way. This time I did not have to implement any confusing movement A.1. All the methods and test for this file were contstructed similiar to these description and instances.

MoveCursor

Here A.1 I made every case correspond to their respective function. For this decided to make a switch statement, such that I could check every case. For instance `InputType.Exit` corresponds to the `Quit` method I made sure to return true for every case expect `PerformMove`.

BoardChecker

The second problem I had to face was within the code was, to Implement the three winning condition: Vertically, Horizontally and Diagonally.

IsRowWin & IsColWin

When I first started implementing `IsRownWin` A.1.1 made sure, to check whether those positions already, were taken. I then used the `Get` method to check whether all three the fields contained a similiar PlayerIdentifier. This method required me to loop through the board's horizontal fields. The `IsColWin` A.1.1 method is very similar to the this, I have just shifted the loops iterative position, such that i get the column fields.

IsDiagWin

For this method A.1.1 I did not use the loops like the previous methods. This is because the positions only lay in two layers rather than three, and they share one; the middle. Therefore I could you the same logic with the `IsTake` & `Get` methods on those particular fields.

CheckBoardState

For this method A.1.1 I were to check three board states: Inconclusive, Tied, Winner.

The first if statement at line 94, checks whether the board is not full with the `IsFull` method and that none of the of the winning conditions have been met.
The second if statement at line 99 checks whether the board is full and that none of the of the winning conditions have been met.
The third else statement gets executed if none of the above if statements are true conditions.

Evaluation

What does all this testing mean and are the results valid? The testing that I did, helped to gain a lot more focus and guided me towards the implemented methods. To disambiguate the ambiguities, I want to talk about the cursor's move methods.

MoveTest

As stated before up and down is not as intuitive as you would think, when it comes to programming. In this example A.2, you get a feel for how the testing was done. The testing checks, if the cursor actually moves as intended. It has been created with the the Arrange, Act, Assert annotation in mind. All of the move functions and tests were created similar to these instances.

This is something that gets eliminated once you start testing the methods. Then you will realize how everything is working. I would also like to mention that my first attempts on creating the win condition methods, had me winning even though PlayerIdentifiers were not the same. This was also quickly excluded from withing the tests. This particular issue was solved by including the `IsTaken` method.

BoardChecker Test

I started by opening the BoardCheckerTest.cs where I had to Implement a few test for the winning condition aswell as test for a tied match and an inconclusive match.

RowWinTest & ColumnWinTest & DiagonalWinTest

I used the some already implemented methods wihtin the Tictactoe program. For the `RowWinTest` I used `TryInsert` for inserting a cross in a horizontally line. This way i could then later Implement the `IsRowWin` A.3 method and test whether it worked or not. I used a very similar method both for `ColumnWinTest` A.3 & `DiagonalWinTest` A.1.1.

InconclusiveTest

For this testA.3 I did make sure to, not fill the board completely with playeridentifiers.

TiedTest

For this test A.3 I made sure, that the board was filled up and that none of the winning conditions were met.

Conclusion

I have now covered the main elements of my third assignment. I have expanded my knowdlegde in object-oriented desgin as well as use of properties and methods. I have implemented the missing parts of TicTacToe and to tested it thorougly. To conclude, I have worked on the incomplete program TicTacToe and made it complete. I did this by focusing on approaching it with a test-driven approach.

A appendix

A.1 Move Methods

```
51      public void MoveUp() {
52          if (Y - 1 >= min)
53          {
54              Y--;
55          }
56
57      }
```

```
62      public void MoveDown() {
63          if (Y + 1 <= max)
64          {
65              Y++;
66          }
67      }
```

```
72      public void MoveLeft() {
73          if (X - 1 >= min)
74          {
75              X--;
76          }
77      }
```

```
82      public void MoveRight() {
83          if (X + 1 <= max)
84          {
85              X++;
86          }
87      }
```

```
106     private bool MoveCursor(InputType inputType) {
107        switch(inputType)
108        {
109             case InputType.Undefined : return true;
110             case InputType.PerformMove : return false;
```

```
111             case InputType.Exit : Quit();
112                                   return true;
113             case InputType.Up : MoveUp();
114                                 return true;
115             case InputType.Down : MoveDown();
116                                   return true;
117             case InputType.Right : MoveRight();
118                                    return true;
119             case InputType.Left : MoveLeft();
120                                   return true;
121             default : return false;
122
123         }
124
125     }
```

A.1.1 BoardChecker Methods

```
22      private bool IsRowWin(Board board) {
23
24          for (int i = 0; i < board.Size; i++)
25          {
26              if (board.IsTaken(0,i) && board.IsTaken(1,i) && board.IsTaken(2,i))
27              {
28                  if (board.Get(0,i) == board.Get(1,i) && board.Get(1,i) == board.Get(2,i))
29                  {
30                      return true;
31                  }
32              }
33          }
34          return false;
35      }
```

```
45      private bool IsColWin(Board board) {
46
47          for (int i = 0; i < board.Size; i++)
48          {
49              if (board.IsTaken(i,0) && board.IsTaken(i,1) && board.IsTaken(i,2))
50              {
51                  if (board.Get(i,0) == board.Get(i,1) && board.Get(i,1) == board.Get(i,2))
52                  {
53                      return true;
54                  }
55              }
```

```
56          }
57          return false;
```

```
68      private bool IsDiagWin(Board board) {
69
70          if (board.IsTaken(0,0) && board.IsTaken(1,1) && board.IsTaken(2,2) ||
71              board.IsTaken(0,2) && board.IsTaken(1,1) && board.IsTaken(2,0))
72          {
73              if (board.Get(0,0) == board.Get(1,1) && board.Get(1,1) == board.Get(2,2))
74              {
75                  return true;
76              }
77
78              else if (board.Get(0,2) == board.Get(1,1) && board.Get(1,1) == board.Get(2,0))
79              {
80                  return true;
81              }
82          }
83          return false;
```

```
92      public BoardState CheckBoardState(Board board) {
93
94      if (!board.IsFull() && !(IsRowWin(board) || IsColWin(board) || IsDiagWin(board)))
95      {
96          return BoardState.Inconclusive;
97      }
98
99      else if(board.IsFull() && !(IsRowWin(board) || IsColWin(board) || IsDiagWin(board)))
100     {
101             return BoardState.Tied;
102     }
103     else
104     {
105         return BoardState.Winner;
106     }
107 }
108 }
```

A.2 CusorTest

```
11      [SetUp]
12      public void Setup() {
```

```
13          var keyToMoveMap = new KeyToMoveMap('i', 'k', 'j', 'l', 'q', ' ');
14          cursor = new Cursor(3, keyToMoveMap);
15          cursor.MoveDown();
16          cursor.MoveRight();
17      }
```

```
24      [Test]
25      public void MoveUpTest() {
26          cursor.MoveUp();
27          var pos = (cursor.position.X, cursor.position.Y);
28          cursor.MoveUp();
29          Assert.True(pos != (pos.Item1+1, pos.Item2+1));
30      }
```

```
32      [Test]
33      public void MoveDownTest() {
34          cursor.MoveDown();
35          var pos = (cursor.position.X, cursor.position.Y);
36          cursor.MoveDown();
37          Assert.True(pos != (pos.Item1+1, pos.Item2+1));    }
```

A.3 BorderCheckerTest

```
17      [Test]
18      public void DiagonalWinTest() {
19          // cross Diagonal win
20          board.TryInsert(0,0,PlayerIdentifier.Cross);
21          board.TryInsert(1,1,PlayerIdentifier.Cross);
22          board.TryInsert(2,2,PlayerIdentifier.Cross);
23          Assert.AreEqual(BoardState.Winner, boardChecker.CheckBoardState(board));
24      }
```

```
26      [Test]
27      public void RowWinTest() {
28          // cross row win
29          board.TryInsert(0,0,PlayerIdentifier.Cross);
30          board.TryInsert(1,0,PlayerIdentifier.Cross);
31          board.TryInsert(2,0,PlayerIdentifier.Cross);
32          Assert.AreEqual(BoardState.Winner, boardChecker.CheckBoardState(board));
33      }
```

```
35      [Test]
36      public void ColumnWinTest() {
37          // cross column win
38          board.TryInsert(0,0,PlayerIdentifier.Cross);
39          board.TryInsert(0,1,PlayerIdentifier.Cross);
40          board.TryInsert(0,2,PlayerIdentifier.Cross);
41          Assert.AreEqual(BoardState.Winner, boardChecker.CheckBoardState(board));
42      }
```

```
44      [Test]
45      public void InconclusiveTest() {
46          // Player Cross
47          board.TryInsert(0,2,PlayerIdentifier.Cross);
48          board.TryInsert(2,0,PlayerIdentifier.Cross);
49
50          // Player Naught
51          board.TryInsert(0,1,PlayerIdentifier.Naught);
52          board.TryInsert(1,0,PlayerIdentifier.Naught);
53
54          Assert.AreEqual(BoardState.Inconclusive, boardChecker.CheckBoardState(board));
55      }
```

```
57      [Test]
58      public void TiedTest() {
59          // Player Cross
60          board.TryInsert(0,0,PlayerIdentifier.Cross);
61          board.TryInsert(0,2,PlayerIdentifier.Cross);
62          board.TryInsert(2,1,PlayerIdentifier.Cross);
63          board.TryInsert(1,2,PlayerIdentifier.Cross);
64          board.TryInsert(2,0,PlayerIdentifier.Cross);
65
66          // Player Naught
67          board.TryInsert(0,1,PlayerIdentifier.Naught);
68          board.TryInsert(1,1,PlayerIdentifier.Naught);
69          board.TryInsert(2,2,PlayerIdentifier.Naught);
70          board.TryInsert(1,0,PlayerIdentifier.Naught);
71
72
73          Assert.AreEqual(BoardState.Tied, boardChecker.CheckBoardState(board));
74      }
75  }
```